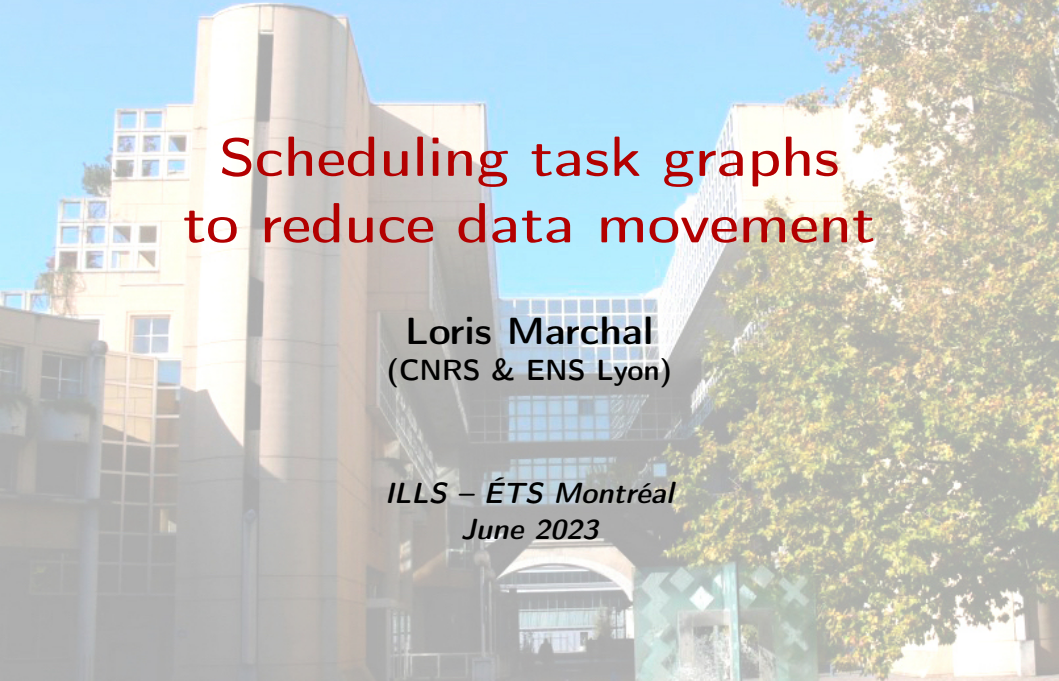
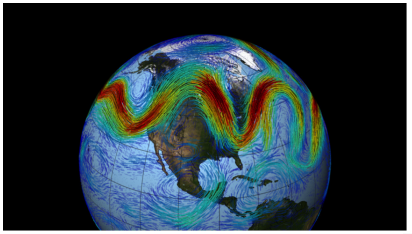


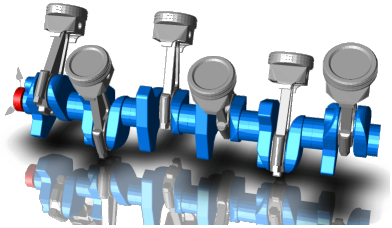
Scheduling task graphs to reduce data movement

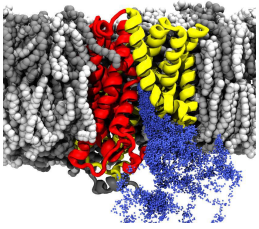
Loris Marchal
(CNRS & ENS Lyon)

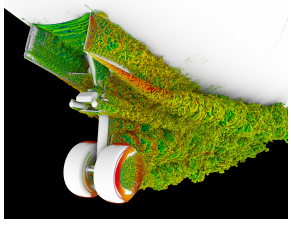
ILLS – ÉTS Montréal
June 2023

High Performance Computing





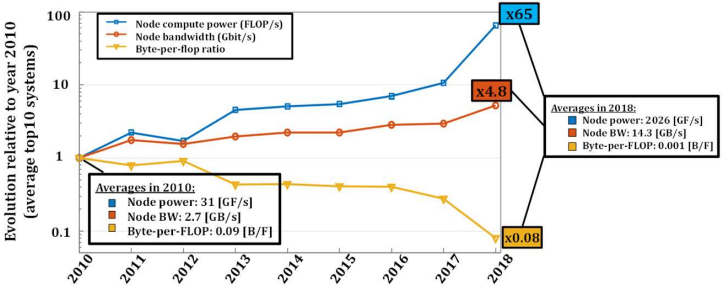




- Numerical simulations drive new discoveries
- Larger systems with better accuracy: more data and computation

Data access problem

Evolution of computing speed vs. data access speed (bandwidth)

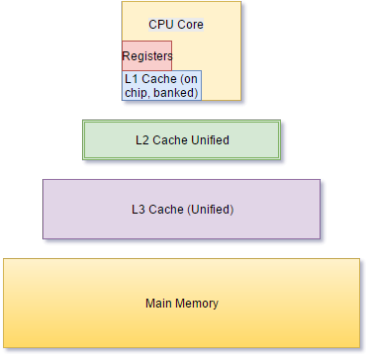


source: https://doi.org/10.1016/B978-0-12-816502-7.00020-8

Byte-per-flop ratio keeps decreasing $\Rightarrow$ Data access critical for performance

Beyond the memory wall

- Time to move the data $>$ Time to compute on the data
- Similar problem in microprocessor design: "memory wall"
- Traditional workaround:
  add a faster but smaller "cache" memory
- Now a hierarchy of caches !



Computing with bounded cache/memory

- Limited amount of fast cache
- Performance sensitive to data locality
- Optimize data reuse
- Avoid data movements (I/Os) between memory and cache(s) (time-consuming and energy-consuming)

In this talk: some algorithmic approaches to this problem

Computing with bounded cache/memory

- Limited amount of fast cache
- Performance sensitive to data locality
- Optimize data reuse
- Avoid data movements (I/Os) between memory and cache(s) (time-consuming and energy-consuming)

In this talk: some algorithmic approaches to this problem

Outline

Task Graph Scheduling and Limited Memory

Pebble game models

Reducing Memory Footprint of Task Graphs

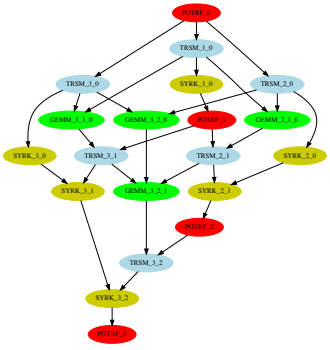
Reducing I/Os for Task Graphs

Outline

Taming HPC platforms with runtime systems

- Write you application as function calls (tasks),
- Specify data input/output (dependencies)
- Provide function codes for specific cores/GPUs
- Let the system do the scheduling at runtime!

```
Cholesky_decomposition(A):
for(k=0; k<N; k++)
   A[k][k]=POTRF(A[k][k])
   for(m=k+1; m<N; m++)
      A[m][k]=TRSM(A[k][k], A[m][k])
   for(n=k+1; n<N; n++)
      A[n][n]=SYRK(A[n][k], A[n][n])
      for(m=n+1; m<N; m++)
         A[m][n]+=GEMM(A[m][k],A[n][k])
```



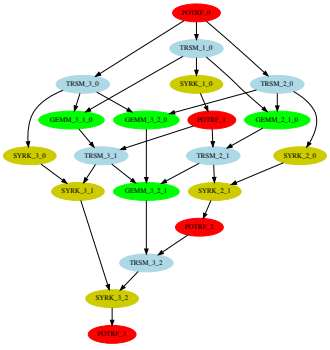
Graph of tasks: Directed Acyclic Graph (DAG)

- Tasks linked with data dependency
- Wide literature on DAG scheduling
- What about memory and data movements (I/Os) ?

Taming HPC platforms with runtime systems

- Write you application as function calls (tasks),
- Specify data input/output (dependencies)
- Provide function codes for specific cores/GPUs
- Let the system do the scheduling at runtime!

```
Cholesky_decomposition(A):
for(k=0; k<N; k++)
   A[k][k]=POTRF(A[k][k])
   for(m=k+1; m<N; m++)
      A[m][k]=TRSM(A[k][k], A[m][k])
   for(n=k+1; n<N; n++)
      A[n][n]=SYRK(A[n][k], A[n][n])
      for(m=n+1; m<N; m++)
         A[m][n]+=GEMM(A[m][k],A[n][k])
```

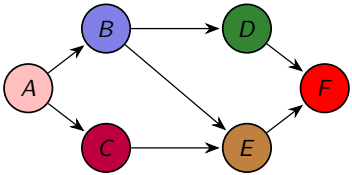


Graph of tasks: Directed Acyclic Graph (DAG)

- Tasks linked with data dependency
- Wide literature on DAG scheduling
- What about memory and data movements (I/Os) ?

Task graph scheduling and memory

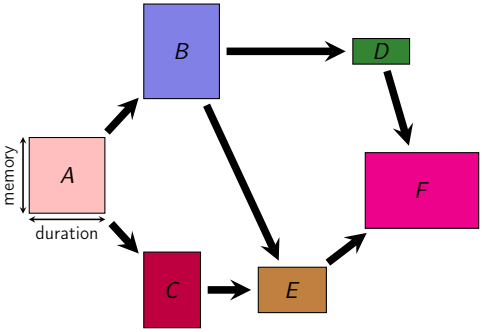
- Consider a simple task graph
- Tasks have durations and memory demands



- Peak memory: maximum memory usage
- Trade-off between peak memory and makespan

Task graph scheduling and memory

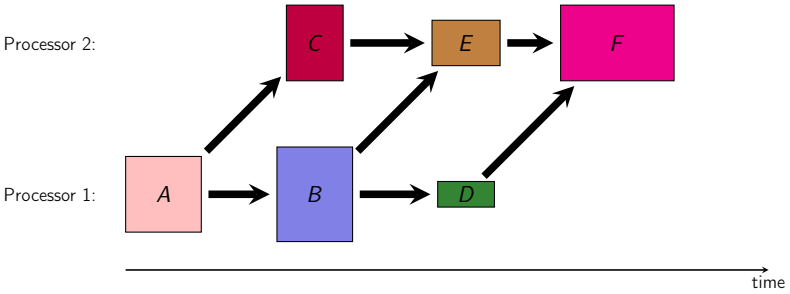
- Consider a simple task graph
- Tasks have durations and memory demands



- Peak memory: maximum memory usage
- Trade-off between peak memory and makespan

Task graph scheduling and memory

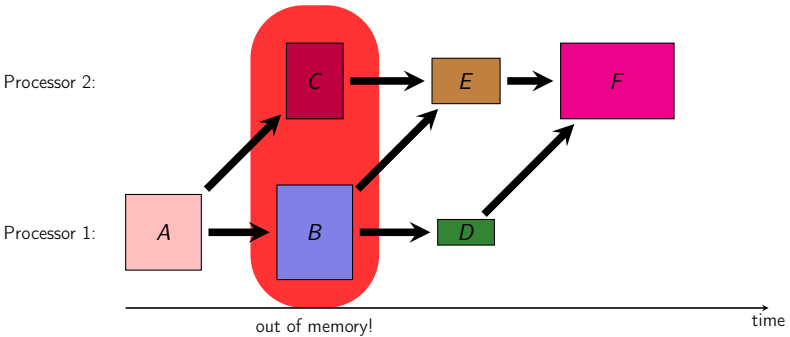
- Consider a simple task graph
- Tasks have durations and memory demands



- Peak memory: maximum memory usage
- Trade-off between peak memory and makespan

Task graph scheduling and memory

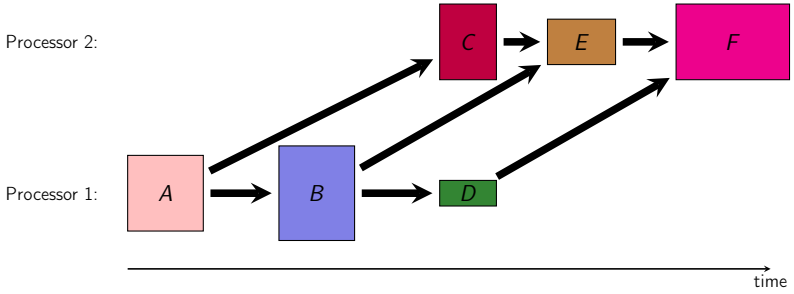
- Consider a simple task graph
- Tasks have durations and memory demands



- Peak memory: maximum memory usage
- Trade-off between peak memory and makespan

Task graph scheduling and memory

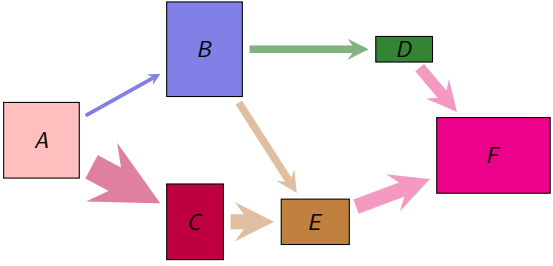
- Consider a simple task graph
- Tasks have durations and memory demands



- Peak memory: maximum memory usage
- Trade-off between peak memory and makespan

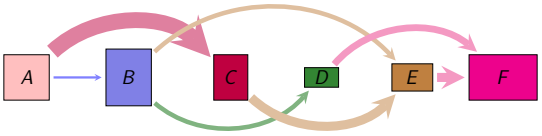
Going back to sequential processing

- Temporary data require memory
- Scheduling influences the peak memory



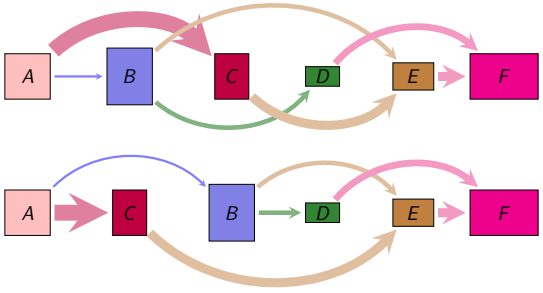
Going back to sequential processing

- Temporary data require memory
- Scheduling influences the peak memory



Going back to sequential processing

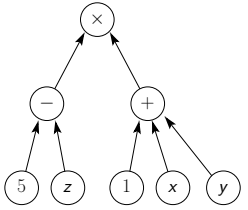
- Temporary data require memory
- Scheduling influences the peak memory



Outline

Pebble game for register allocation

- From the 70s: limit usage of scarce registers
- Model expressions as Directed Acyclic Graphs



$(5 - z) \times (1 + x + y)$

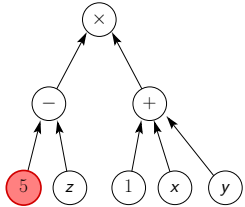
Rules of the game:

- A pebble may be placed on a source node at any time (LOAD)
- If all predecessors of v are pebbled, a pebble may be placed on v. (COMPUTE)
- A pebble may be removed from a vertex at any time. (EVICT)
- Goal: computation all vertices, use minimal number of pebbles

Results: Optimal algorithms for trees — NP-hard on general DAGs

Pebble game for register allocation

- From the 70s: limit usage of scarce registers
- Model expressions as Directed Acyclic Graphs



$(5 - z) \times (1 + x + y)$

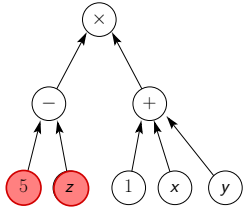
Rules of the game:

- A pebble may be placed on a source node at any time (LOAD)
- If all predecessors of v are pebbled, a pebble may be placed on v. (COMPUTE)
- A pebble may be removed from a vertex at any time. (EVICT)
- Goal: computation all vertices, use minimal number of pebbles

Results: Optimal algorithms for trees — NP-hard on general DAGs

Pebble game for register allocation

- From the 70s: limit usage of scarce registers
- Model expressions as Directed Acyclic Graphs



$(5 - z) \times (1 + x + y)$

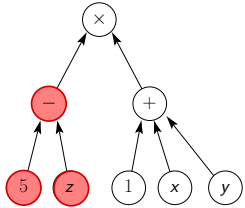
Rules of the game:

- A pebble may be placed on a source node at any time (LOAD)
- If all predecessors of v are pebbled, a pebble may be placed on v. (COMPUTE)
- A pebble may be removed from a vertex at any time. (EVICT)
- Goal: computation all vertices, use minimal number of pebbles

Results: Optimal algorithms for trees — NP-hard on general DAGs

Pebble game for register allocation

- From the 70s: limit usage of scarce registers
- Model expressions as Directed Acyclic Graphs



$(5 - z) \times (1 + x + y)$

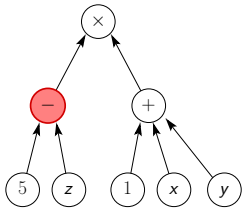
Rules of the game:

- A pebble may be placed on a source node at any time (LOAD)
- If all predecessors of v are pebbled, a pebble may be placed on v. (COMPUTE)
- A pebble may be removed from a vertex at any time. (EVICT)
- Goal: computation all vertices, use minimal number of pebbles

Results: Optimal algorithms for trees — NP-hard on general DAGs

Pebble game for register allocation

- From the 70s: limit usage of scarce registers
- Model expressions as Directed Acyclic Graphs



$(5 - z) \times (1 + x + y)$

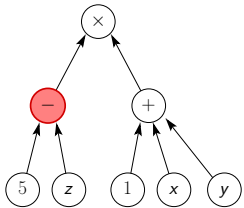
Rules of the game:

- A pebble may be placed on a source node at any time (LOAD)
- If all predecessors of v are pebbled, a pebble may be placed on v. (COMPUTE)
- A pebble may be removed from a vertex at any time. (EVICT)
- Goal: computation all vertices, use minimal number of pebbles

Results: Optimal algorithms for trees — NP-hard on general DAGs

Pebble game for register allocation

- From the 70s: limit usage of scarce registers
- Model expressions as Directed Acyclic Graphs



$(5 - z) \times (1 + x + y)$

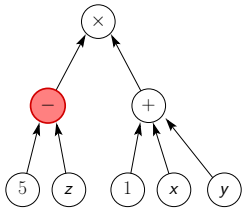
Rules of the game:

- A pebble may be placed on a source node at any time (LOAD)
- If all predecessors of v are pebbled, a pebble may be placed on v. (COMPUTE)
- A pebble may be removed from a vertex at any time. (EVICT)
- Goal: computation all vertices, use minimal number of pebbles

Results: Optimal algorithms for trees — NP-hard on general DAGs

Pebble game for register allocation

- From the 70s: limit usage of scarce registers
- Model expressions as Directed Acyclic Graphs



$(5 - z) \times (1 + x + y)$

Rules of the game:

- A pebble may be placed on a source node at any time (LOAD)
- If all predecessors of v are pebbled, a pebble may be placed on v. (COMPUTE)
- A pebble may be removed from a vertex at any time. (EVICT)
- Goal: computation all vertices, use minimal number of pebbles

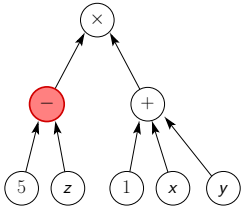
Results: Optimal algorithms for trees — NP-hard on general DAGs

When memory too limited: minimize I/Os

Red/Blue pebble game [Hong & Kung, 1981]

New rules:

- Limited number of red pebbles (=memory slots)
- Replace red pebble by blue pebble (WRITE)
- Replace blue pebble by red pebble (READ)



Goal: minimize number of WRITE

Model applies to any two-memory system:
(fast, bounded) memory vs. (slow, large) disk

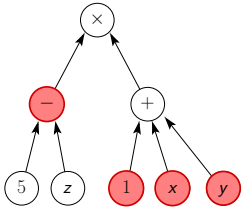
- Successful to design lower bounds on I/Os and optimal algorithms
- Basis for other studies: communication-avoiding algorithms
  (recomputations may be allowed or forbidden)

When memory too limited: minimize I/Os

Red/Blue pebble game [Hong & Kung, 1981]

New rules:

- Limited number of red pebbles (=memory slots)
- Replace red pebble by blue pebble (WRITE)
- Replace blue pebble by red pebble (READ)



Goal: minimize number of WRITE

Model applies to any two-memory system:
(fast, bounded) memory vs. (slow, large) disk

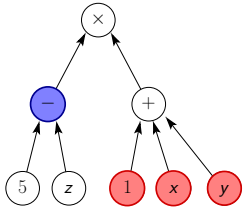
- Successful to design lower bounds on I/Os and optimal algorithms
- Basis for other studies: communication-avoiding algorithms
  (recomputations may be allowed or forbidden)

When memory too limited: minimize I/Os

Red/Blue pebble game [Hong & Kung, 1981]

New rules:

- Limited number of red pebbles (=memory slots)
- Replace red pebble by blue pebble (WRITE)
- Replace blue pebble by red pebble (READ)



Goal: minimize number of WRITE

Model applies to any two-memory system:
(fast, bounded) memory vs. (slow, large) disk

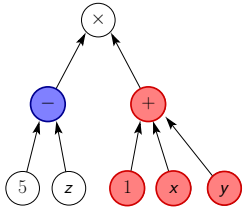
- Successful to design lower bounds on I/Os and optimal algorithms
- Basis for other studies: communication-avoiding algorithms
  (recomputations may be allowed or forbidden)

When memory too limited: minimize I/Os

Red/Blue pebble game [Hong & Kung, 1981]

New rules:

- Limited number of red pebbles (=memory slots)
- Replace red pebble by blue pebble (WRITE)
- Replace blue pebble by red pebble (READ)



Goal: minimize number of WRITE

Model applies to any two-memory system:
(fast, bounded) memory vs. (slow, large) disk

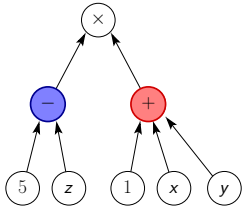
- Successful to design **lower bounds on I/Os** and optimal algorithms
- Basis for other studies: communication-avoiding algorithms
  (recomputations may be allowed or forbidden)

When memory too limited: minimize I/Os

Red/Blue pebble game [Hong & Kung, 1981]

New rules:

- Limited number of red pebbles (=memory slots)
- Replace red pebble by blue pebble (WRITE)
- Replace blue pebble by red pebble (READ)



Goal: minimize number of WRITE

Model applies to any two-memory system:
(fast, bounded) memory vs. (slow, large) disk

- Successful to design **lower bounds on I/Os** and optimal algorithms
- Basis for other studies: communication-avoiding algorithms
  (recomputations may be allowed or forbidden)

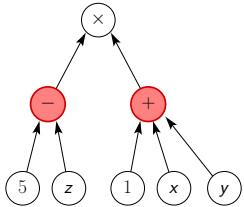
When memory too limited: minimize I/Os

Red/Blue pebble game [Hong & Kung, 1981]

New rules:

- Limited number of red pebbles (=memory slots)
- Replace red pebble by blue pebble (WRITE)
- Replace blue pebble by red pebble (READ)

Goal: minimize number of WRITE



Model applies to any two-memory system:
(fast, bounded) memory vs. (slow, large) disk

- Successful to design **lower bounds on I/Os** and optimal algorithms
- Basis for other studies: communication-avoiding algorithms
  (recomputations may be allowed or forbidden)

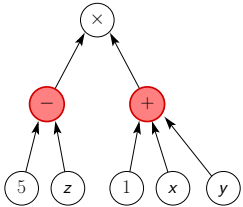
When memory too limited: minimize I/Os

Red/Blue pebble game [Hong & Kung, 1981]

New rules:

- Limited number of red pebbles (=memory slots)
- Replace red pebble by blue pebble (WRITE)
- Replace blue pebble by red pebble (READ)

Goal: minimize number of WRITE



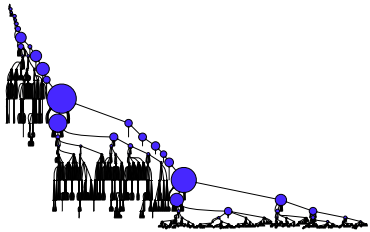
Model applies to any two-memory system:
(fast, bounded) memory vs. (slow, large) disk

- Successful to design **lower bounds on I/Os** and optimal algorithms
- Basis for other studies: communication-avoiding algorithms (recomputations may be allowed or forbidden)

Outline

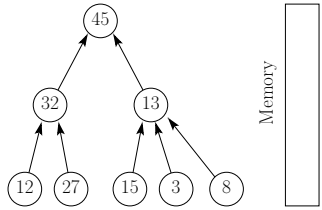
Generalized (Black) Pebble Game

- Sparse matrix factorization
- Task graph: tree (with dependencies towards the root)
- Large temporary data



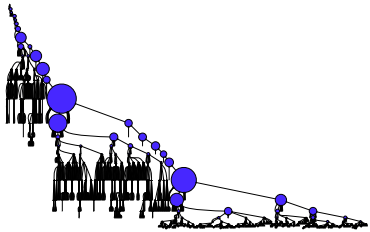
Generalized pebble game [Liu 1986]:

- Node have heterogeneous weights (memory demand)
- Compute task = replace inputs by outputs in memory
- output memory $\neq \sum$ input memory



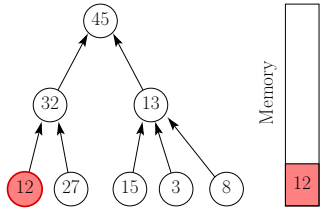
Generalized (Black) Pebble Game

- Sparse matrix factorization
- Task graph: tree (with dependencies towards the root)
- Large temporary data



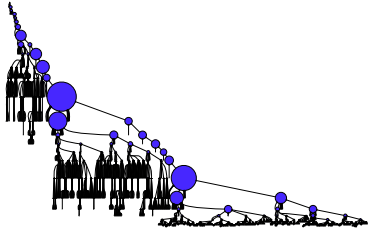
Generalized pebble game [Liu 1986]:

- Node have heterogeneous weights (memory demand)
- Compute task = replace inputs by outputs in memory
- output memory $\neq \sum$ input memory



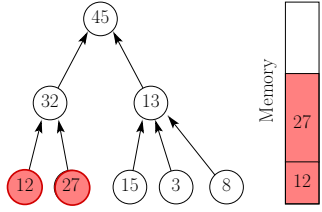
Generalized (Black) Pebble Game

- Sparse matrix factorization
- Task graph: tree (with dependencies towards the root)
- Large temporary data



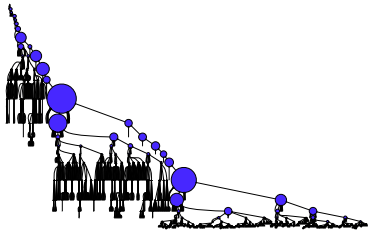
Generalized pebble game [Liu 1986]:

- Node have heterogeneous weights (memory demand)
- Compute task = replace inputs by outputs in memory
- output memory $\neq \sum$ input memory



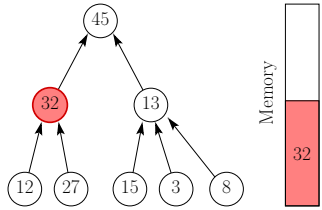
Generalized (Black) Pebble Game

- Sparse matrix factorization
- Task graph: tree (with dependencies towards the root)
- Large temporary data



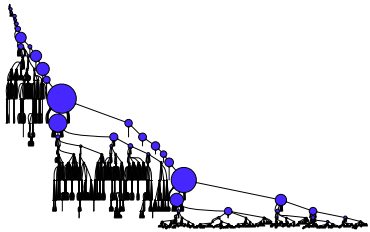
Generalized pebble game [Liu 1986]:

- Node have heterogeneous weights (memory demand)
- Compute task = replace inputs by outputs in memory
- output memory $\neq \sum$ input memory



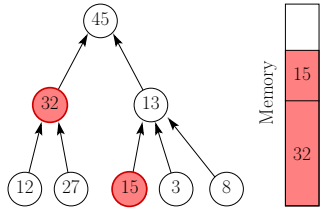
Generalized (Black) Pebble Game

- Sparse matrix factorization
- Task graph: tree (with dependencies towards the root)
- Large temporary data



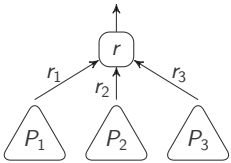
Generalized pebble game [Liu 1986]:

- Node have heterogeneous weights (memory demand)
- Compute task = replace inputs by outputs in memory
- output memory $\neq \sum$ input memory



Tree Traversals with Smallest Memory [Liu 1987]

1. Restrict on postorder traversals: simpler control



- Complete subtree one after the other
- P_i: memory peak when processing subtree i
- r_i: residual memory after processing subtree i

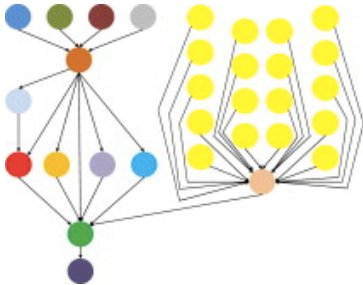
- For a given traversal, memory peak of the subtree:
$$\max\left\{P_1,\ r_1+P_2,\ r_1+r_2+P_3,\ Mem(r)\right\}$$
- Peak minimized when subtrees are sorted by decreasing $P_i - r_i$

2. Optimal tree traversal for memory (not necessarily postorder)

- Possibly switch from one subtree to another
- Same intuition, slightly more complex algorithm, complex proof

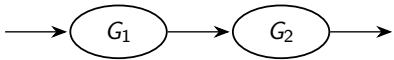
Minimizing memory for series-parallel graphs (1/3)

- Not all task graphs are trees
- But many exhibit regularities
- Important subclass: SP graphs

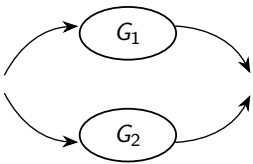


single vertex:

series composition:

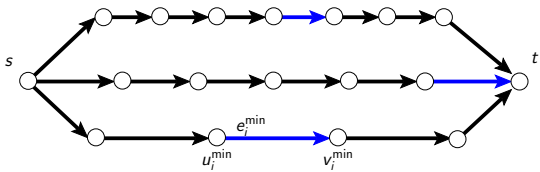


parallel composition:



Minimizing memory for series-parallel graphs (2/3)

Base case: **parallel chains**:



Edge using the minimum amount of memory, on each chain: $e_1, \ldots, e_n$.

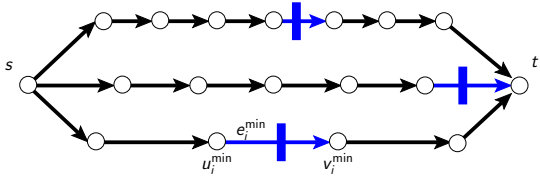
Lemma

There exists an schedule with minimal memory stopping on edges $e_1, \ldots, e_n$.

1. Split the graph on minimal cut $e_1, \ldots, e_n$
2. Apply Liu's algorithm on resulting trees

Minimizing memory for series-parallel graphs (2/3)

Base case: **parallel chains**:



Edge using the minimum amount of memory, on each chain: $e_1, \ldots, e_n$.

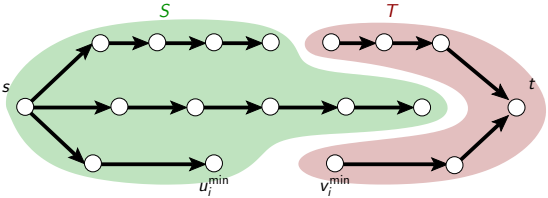
Lemma

There exists an schedule with minimal memory stopping on edges $e_1, \ldots, e_n$.

1. Split the graph on minimal cut $e_1, \ldots, e_n$
2. Apply Liu's algorithm on resulting trees

Minimizing memory for series-parallel graphs (2/3)

Base case: **parallel chains**:



Edge using the minimum amount of memory, on each chain: $e_1, \ldots, e_n$.

Lemma

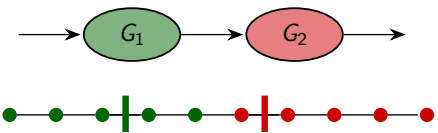
There exists an schedule with minimal memory stopping on edges $e_1, \ldots, e_n$.

1. Split the graph on minimal cut $e_1, \ldots, e_n$
2. Apply Liu's algorithm on resulting trees

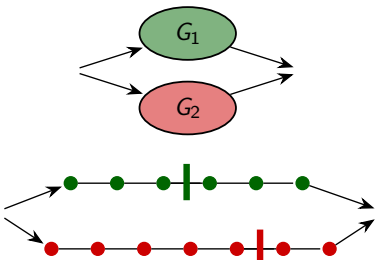
Minimizing memory for series-parallel graphs (3/3)

- Follow recursive definition of the graph
- Simultaneously compute minimal cut and optimal schedule
- Replace subgraph by linear chain corresponding to the schedule

series composition:



parallel composition:



Heuristic method for general graphs

- Transform graph into SP-graph by adding synchronisation points
- Compute optimal schedule on obtained SP-graph

Other results on task graph scheduling for memory

Parallel processing on shared-memory platforms

- Tradeoff between time-to-solution and memory
- Dynamique scheduling under memory constraint
  - Dynamic scheduling may go out of memory
  - Transform the graph (add specific edges)
  - Guarantee memory stays below specified threshold

Algorithms implemented in `gitlab.inria.fr/lmarchal/memdag`

Application for DNN inference


Fused Depthwise Tiling for Memory Optimization in TinyML Deep Neural Network Inference

Rafael Stahl, Daniel Mueller-Gritschneder, Ulf Schlichtmann

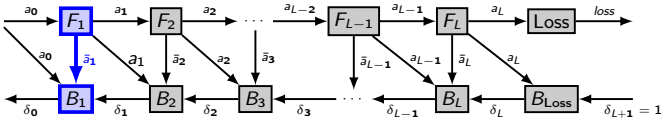
Fused Depthwise Tiling for Memory Optimization in TinyML Deep Neural Network Inference
Rafael Stahl, Daniel Mueller-Gritschneder, Ulf Schlichtmann
tinyML Research Symposium 2023

arXiv:2303.17878

- Inference of DNN on microcontrollers
- Model tiling to reduce memory
- Chain graph between NN layers $\rightarrow$ Series-Parallel graph
- Use of our optimal algorithm for SP-graphs

Application for DNN training

Work by Lionel Eyraud-Dubois, Olivier Beaumont et. al.



- DNN: specific graph ($\approx$ double chain)
- Huge memory demand (store activations)
- Delete/recompute some activations (rematerialization)
- Offload some activations on slow storage

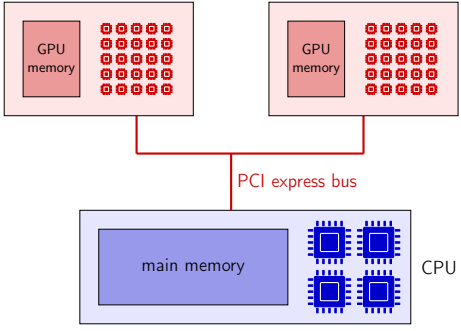
Design of efficient rematerialization/offloading strategies:

`https://gitlab.inria.fr/hiepacs/rotor`

Outline

Platform model

- Memory too scarce to accomodate all (input) data
- Data initially on a large, slow storage



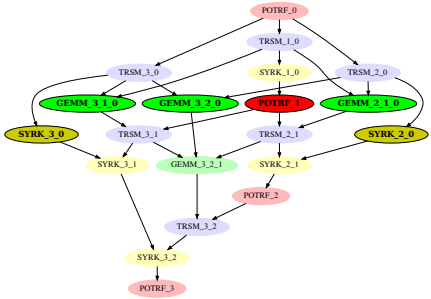
GPUs provide large speed-ups for reduced energy, but:

- limited memory within GPU
- connected through bus with limited bandwidth

Dynamic view of a task graph

At any time step: consider only available tasks

- Independant tasks
- Sharing some input data

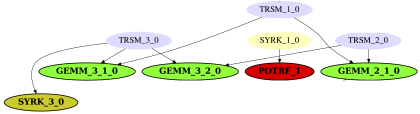


→ bipartite graph between data and tasks

Dynamic view of a task graph

At any time step: consider only available tasks

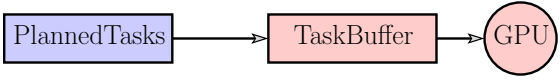
- Independant tasks
- Sharing some input data



$\rightarrow$ bipartite graph between data and tasks

Dynamic scheduling of task graphs

- Tasks appear over time (task graph discovered at runtime)
- Two questions:
  - Partition tasks among GPUs
  - Order task on each GPUs
- When task input data not on GPU: load it from main memory (possibly before the execution: prefetching)
- When memory is full: evict data Eviction policy

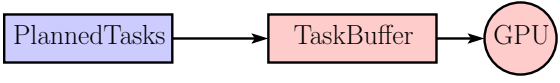


Two sorted sets of tasks per GPU (FIFO):

1. TaskBuffer: tasks definitively allocated on a GPU (data possibly being prefetched)
2. PlannedTasks: good candidate tasks for a GPU

Dynamic scheduling of task graphs

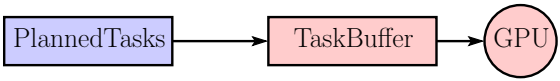
- Tasks appear over time (task graph discovered at runtime)
- Two questions:
  - Partition tasks among GPUs
  - Order task on each GPUs
- When task input data not on GPU: load it from main memory (possibly before the execution: prefetching)
- When memory is full: evict data Eviction policy



Two sorted sets of tasks per GPU (FIFO):

1. TaskBuffer: tasks definitively allocated on a GPU (data possibly being prefetched)
2. PlannedTasks: good candidate tasks for a GPU

DARTS (Data-Aware Reactive Task Scheduling)

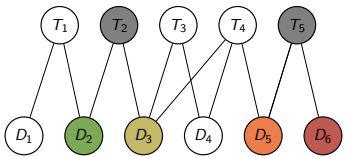


How to fill PlannedTasks$_k$ when needed:

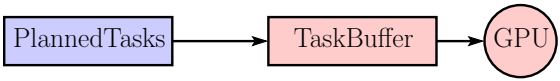
1. Concentrate on data, choose "best" data to load
2. Look for tasks that GPU_k can do with D + its current data
3. Choose data with largest ratio:

$$\frac{\text{computation time of tasks enabled with } D}{\text{time needed to transfer data } D}$$

4. Break ties with task priorities (critical path)
5. Put all "enabled" tasks in PlannedTasks$_k$



DARTS (Data-Aware Reactive Task Scheduling)

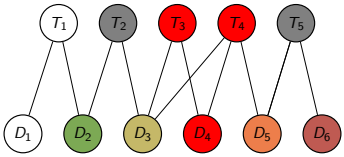


How to fill PlannedTasks$_k$ when needed:

1. Concentrate on data, choose "best" data to load
2. Look for tasks that GPU_k can do with D + its current data
3. Choose data with largest ratio:

$$\frac{\text{computation time of tasks enabled with } D}{\text{time needed to transfer data } D}$$

4. Break ties with task priorities (critical path)
5. Put all "enabled" tasks in PlannedTasks$_k$



Custom eviction policy

Existing cache management policies:

- With no information about future tasks/requests:
  simple policies based on past usage, eg. Last Recently Used (LRU)
- With perfect information on future accesses:
  Belady's rule (1966): evict data with furthest access

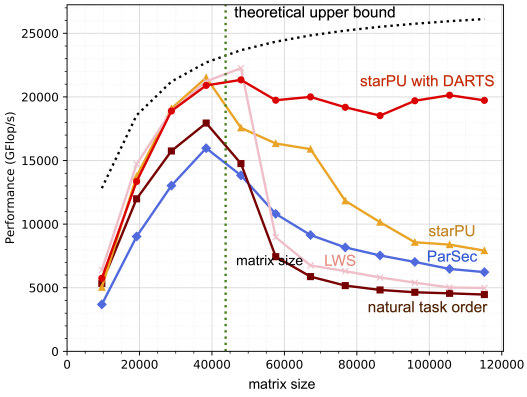
In our system:

- No complete vision of the future ☹
- Window of allocated tasks and planned tasks ☺

Eviction policy for DARTS:

1. Remove data used by fewest tasks in PlannedTasks
2. If needed, apply Belady's rule on TaskBuffer

Performance on memory-limited GPUs



- Cholesky factorization on 2 GPUs
- Green vertical line: matrix uses all available memory

Conclusion

- Concentrate on data movements is the key for performance
- Algorithm design can help re-organizing computations for better data reuse
- With help from: compiler theory, cache management, ...
- Scheduling of task graphs: powerful model with applications for linear algebra, DNN training, workflows,...